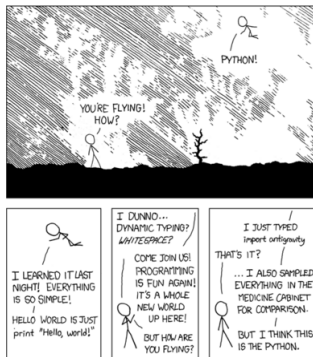


## Qui veut du Python ?

Anthony VEREZ (netantho@minet.net)

Décembre 2011

# Pourquoi Python ?



## Quelques informations

- Langage interprété, interpréteur interactif avec la commande *python*
- 8<sup>ème</sup> langage le plus utilisé
- Utilisé par la NASA, Google, Battlefield 2, ...
- Deux branches : 2.x (actuellement 2.7.2) depuis 2001 et 3.x (actuellement 3.2.2) depuis 2009. Quand ce n'est pas précisé, je vais travailler ici avec **Python 2.x**
- Indentation des blocs avec espaces obligatoire
- Librairie standard riche
- Programmation orienté objet
- Python est déjà installé sur la plupart des distributions Linux ;)

# Indentation en C

Lâchez le troll !

```
1 #include <stdio.h>
2
3 /* Donne FALSE */
4 int main()
5 {
6     if (1)
7         if (0)
8             printf("TRUE\n");
9     else
10         printf("FALSE\n");
11     return 0;
12 }
```

# Indentation en Python

C'est mieux ! :P

```
1  foo = True
2  str = "foo"
3  if foo:
4      if str == "bar":
5          print "Bar"
6      else:
7          print "Foo"
8          print "Foo !!!"
9  else:
10     print "Bar !!!"
```

# Typage Fort

Python est un langage à typage fort dynamique.

- Les types sont déterminés à l'exécution
- La compilation et l'exécution détectent les erreurs de typage
- Pas de conversion implicite de types, sauf un peu pour les nombres (!= PHP)

## Je veux de la doc !

- Documentation officielle
- Plongez au cœur de Python
- AFPY Association Francophone Python
- Présentations de Boris Feld, cette présentation en est inspirée
- Les bonnes pratiques
- PEP Python Enhancement Proposals

```
1 >>> help(fonction)
2 >>> object.__dict__
```

## Types de variables

```
1 >>> type(42)
2 <type 'int'>
3 >>> type(42.) #42.==42.0
4 <type 'float'>
5 >>> type("texte")
6 <type 'str'>
7 >>> type(True)
8 <type 'bool'>
9 >>> type(0L)
10 <type 'long'>
11 >>> type(1+1j)
12 <type 'complex'>
```

Pas d'équivalent de *char*, on n'a que des chaînes de caractères

```
1 >>> type('a')
2 <type 'str'>
```

Conversion de types (typecasting) avec *int(foo)*, *str(foo)*, ...



## Chaînes de caractères

- Concaténation avec `+` ou `join(liste)`

```
1 >>> "foo "+str(42)+" bar"
2 'foo 42 bar'
3 >>> couleurs = ['rouge', 'bleu', 'vert', 'jaune']
4 >>> ' '.join(couleurs) # pour liste
5 'rouge bleu vert jaune'
6 >>> x = 'string sur plusieurs \
7 >>> ... lignes'
8 >>> print x
9 'string sur plusieurs lignes'
10 >>> print "Vive\nINTech"
11 Vive
12 INTech
```

## "Printf"

```
1 >>> print "%.5d euros" % (3*10)
2 00030 euros
3 >>> print "%d + %d = %d" % (2, 3, (2+3))
4 2 + 3 = 5
```

## Opérations arithmétiques

- Opérateurs :  $+$   $-$   $*$   $/$   $\%$
- Une opération sur deux nombres dans deux ensembles différents donne le résultat dans le plus "grand" (au sens de l'inclusion) ensemble

```
1 >>> 2*5.  
2 10.0  
3 >>> 2*5j  
4 10j
```

# Affectations

```
1 >>> x = 0
2 >>> x
3 0
4 >>> y, z = 1, 0
5 >>> y
6 1
7 >>> z
8 0
```

# Booléens

Valeurs considérées comme fausses :

- False
- None
- 0, 0.0, 0L, 0j
- ""
- (), [], { } on en reparle ;)

```
1 >>> x or y # OU logique (inclusif)
2 >>> x and y # ET logique
3 >>> not x # NON logique
```

# Listes

Structure ordonnée de taille dynamique contenant une collection d'objets de types qui peuvent être différents

```
1 >>> x = []
2 >>> type(x)
3 <type 'list'>
4 >>> x = [1, 2, 'a', 'c']
5 >>> x
6 [1, 2, 'a', 'c']
7 >>> x[3]
8 'c'
9 >>> list('Plop')
10 ['P', 'l', 'o', 'p']
```

## Listes - Fonctions utiles

```
1 >>> x.append(5)
2 >>> x
3 [1, 2, 'a', 'c', 5]
4 >>> x.remove('a')
5 >>> x
6 [1, 2, 'c', 5]
7 >>> len(x) # taille
8 4
9 >>> l = [1, 2, 1, 1]
10 >>> l.count(1)
11 3
```

```
12 >>> l.count(2)
13 1
14 >>> l[1:]
15 [2, 1, 1]
16 >>> l[::-1]
17 [1, 1, 2, 1]
18 >>> # Premier indice
19 >>> # d'un element
20 >>> l.index(1)
21 0
```

# Tuples

Comparables aux listes à la différence que les valeurs des tuples ne sont pas modifiables

```
1 >>> x = (1,2)    11 (2,)    21 1
2 >>> x            12 >>> # pas tuple 22 >>> x[0] = 2
3 (1, 2)           13 >>> x = (1) 23 Traceback (most
4 >>> type(x)       14 >>> x     24 recent call last):
5 <type 'tuple'>    15 1       25 File "<stdin>",
6 >>> x[1]          16 >>> # tuple 26 line 1, in <module>
7 2                17 >>> x = (1,) 27 TypeError: 'tuple'
8 >>> x[:1]         18 >>> x     28 object does not
9 (1,)             19 (1,)    29 support item
10 >>> x[1::2]       20 >>> x[0] 30 assignment
```



## Tuples - Fonctions utiles

```
1 >>> l = (1, 2, 2, 3, 3, 3) >>> l.index(1)
2 >>> l.count(2)                8 0
3 2                               9 >>> l.index(2)
4 >>> l.count(3)                10 1
5 3                               11 >>> len(l)
6 >>> l = (1, 2, 1)             12 3
```

## Indexation négative

```
1 >>> x = [1, 2, 3, 4, 5]
2 >>> x[-1]
3 5
4 >>> x[-2]
5 4
6 >>> x[-5]
7 1
```

# Dictionnaires

Accès aux éléments non pas par des index mais par des clés (chaîne de caractères, tuple, ...), genre de "tableau associatif".

- Clés non modifiables
- Structure non ordonnée
- Une clé pointe vers une seule valeur (qui peut néanmoins être une liste, un tuple ou un dictionnaire)

## Dictionnaires

```
1 >>> positions = {}
2 >>> positions
3 {}
4 >>> positions = {(0,0): True, (0,1): False}
5 >>> positions
6 {(0, 1): False, (0, 0): True}
7 >>> positions[(0,0)]
8 True
9 >>> positions[(1,1)] = 'a'
10 >>> positions
11 {(0, 1): False, (0, 0): True, (1, 1): 'a'}
12 >>> positions[(0,1)] = True
13 >>> positions
14 {(0, 1): True, (0, 0): True, (1, 1): 'a'}
```

## Dictionnaires - Fonctions utiles

```
1 >>> len(positions)
2 3
3 >>> positions.values()
4 [True, True, False]
5 >>> positions.keys()
6 [(0, 1), (0, 0), (1, 1)]
7 >>> del positions[(0,1)]
8 >>> positions
9 {(0, 0): True, (1, 1): 'a'}
10 >>> (1,1) in positions
11 True
```

# Les itérables

Sont itérables

- Les chaînes de caractères
- Les tuples, listes et dictionnaires
- Les fichiers (sur les lignes)

**Attention**, sur les dictionnaires, les éléments itérés sont les clés alors que pour listes et tuples ce sont les valeurs

## Les itérables

```
1 >>> ch = 'coucou'
2 >>> 'o' in ch
3 True
4 >>> map(lambda x: x+'a', ch)
5 ['ca', 'oa', 'ua', 'ca', 'oa', 'ua']
6 >>> dict = {'c': 'b', 'b': 'c', 'a': 'd'}
7 >>> sorted(dict)
8 ['a', 'b', 'c']
9 >>> sorted(dict.values())
10 ['b', 'c', 'd']
11 >>> reversed(ch)
12 <reversed object at 0x1800a90>
13 >>> ''.join(list(reversed(ch))) # ne caste pas en str
14 'uocuoC'
```

# Conditions

## Les opérateurs de comparaison

- ==
- !=
- <=, >=, <, >

```
1 >>> x = 0
2 >>> if x > 0:
3 ...     print "Plus grand que 0"
4 ...     elif x < 0:
5 ...     print "Plus petit que 0"
6 ...     else:
7 ...     print "Egal 0"
8 ...
9 Egal 0
```



## Boucle for

- Sur itérable

```

1  >>> x = (1, 2, 3)          5  1
2  >>> for i in x:            6  2
3  ...     print i           7  3
4  ...

```

- utilisation "Classique"

```

1  >>> range(5)                7  ...
2  [0, 1, 2, 3, 4]             8  0 1 2 3 4
3  >>> range(2, 7)             9  >>> for i in range(5,2,-1):
4  [2, 3, 4, 5, 6]            10 ...     print i,
5  >>> for i in range(5):11 ...
6  ...     print i,           12 5 4 3

```

## Boucle while

```
1 >>> b, i = True, 0
2 >>> while b:
3     ...     print i,
4     ...     i += 1
5     ...     if i >= 5:
6     ...         b = not b
7     ...
8 0 1 2 3 4
```

## break et continue

```
1 >>> compteur = 0
2 >>> while True:
3     ...     print compteur ,
4     ...     compteur += 1
5     ...     if compteur >= 5:
6     ...         break # Arrête l'exécution de la boucle
7     ...
8 0 1 2 3 4
9 >>> for x in range(10):
10    ...     if x % 2 == 0:
11    ...         continue # Passe au bloc suivant
12    ...     print x
13    ...
14 1 3 5 7 9
```

## Entrées utilisateur

```
1 >>> s = raw_input('Vous suivez encore ? ')
2 Vous suivez encore ? oui
3 >>> s
4 'oui'
```

```
1 $ echo "import sys" >> cli.py
2 $ echo "print sys.argv" >> cli.py
3 $ python cli.py INTech Powa
4 ['cli.py', 'INTech', 'Powa']
```

# Fonctions

```
1 >>> def hello_world():
2     ...     print "Hello World!"
3     ...
4 >>> hello_world()
5 Hello World!
6 >>> def fonction(arg1, arg2, arg3 = "val3", arg4 = "val4")
7     ...     print arg1, arg2, arg3, arg4
8     ...
9 >>> fonction("val1", "val2")
10 val1 val2 val3 val4
11 >>> fonction(arg2 = "val1", arg1 = "val2")
12 val2 val1 val3 val4
13 >>> fonction("val1", "val2", arg4 = "val1")
14 val1 val2 val3 val1
```

## Démo !!!

### Challenge

*Afficher les 10 derniers tweets contenant 'python' avec l'utilisateur associé et la date*

## Corrigé

```
1 import urllib
2
3 url = urllib.urlopen("http://search.twitter.com/search.json?q=python")
4
5 import json
6
7 monjson = json.loads(url.read())
8 for elem in monjson['results']:
9     print elem['from_user_name']+" le "+elem['created_at']
10    print elem['text']+"\n"
```

Tadam ! Rapide, non ?

## Les fichiers

```
1 >>> f = open('fichier', 'w')
2 >>> f.write("Coucou")
3 >>> f.close()
4
5 >>> # Permet de fermer le fichier automatiquement
6 >>> # apres traitements
7 >>> with open('fichier', 'r') as f:
8     ...     print f.readlines()
9     ['Quoi']
10 >>> with open('fichier', 'r') as f:
11     ...     print f.read()
12     'Quoi'
```

Pour les chemins, voir le module os



## Idées principales et jargon

Un *objet* en programmation est une structure de données qui contient à la fois :

- Des *attributs* : données sous forme de variables
  - de classe : propres à une classes et communs à tous les objets de même type
  - d'instance : propres à un objet
- Des *méthodes* : des moyens de traiter ces données sous forme de fonctions

Un objet est une instance d'une *classe*

Le *constructeur* est la méthode qui est appelée à la création d'un objet

# Classes

```
1  >>> class Robot:
2  ...     def __init__(self, couleur, roues): # constructeur
3  ...         self.couleur = couleur # attribut d'instance
4  ...         Robot.roues = roues # attribut de classe
5  ...     def changerCouleur(self, couleur):
6  ...         self.couleur = couleur
7  ...
>>> bot1=Robot("rouge",2)           'jaune'
>>> bot1.roues                       >>> bot2=Robot("bleu",3)
2                                   >>> bot1.roues
>>> bot1.couleur                     3
'rouge'                             >>> bot2.couleur
>>> bot1.changerCouleur("jaune")'bleu'
>>> bot1.couleur
```

# Héritage

```
1 >>> class A:
2     ...     a = 'A'
3 >>> class B:
4     ...     b = 'B'
5     ...
6 >>> # la classe C herite des classes A et B
7 >>> class C(A,B):
8     ...     pass
9 >>> c = C
10 >>> c.a
11 'A'
```

On peut aussi hériter de méthodes de la même façon

## Surcharge d'opérateurs

Définir les opérateurs entre des objets de même types. Ils faut définir des méthodes spéciales dans les classes :

- `object.__lt__(self, other) : <`
- `object.__le__(self, other) : <=`
- `object.__eq__(self, other) : ==`
- `object.__ne__(self, other) : !=`
- `object.__gt__(self, other) : >`
- `object.__ge__(self, other) : >=`

## Modularité

Un programme python peut s'organiser sous forme de modules.

Créons le module script :

script.py

```
1 var = 3
2 def ma_fonction():
3     print "ma fonction"
4 class C:
5     pass
6 print "mon module"
```

Dans le même dossier, lançons l'interpréteur interactif :

```
1 >>> import script # module execute a l'importation
2 mon module
3 >>> locals()
4 {'__builtins__': <module '__builtin__' (built-in)>, '__name__':
5   '__main__', 'script': <module 'script' from 'script.py'>}
```

## Modularité

Pour éviter ce genre de chose, on peut faire ça :  
modularite\_execution.py

```
1 print("Nom : " + __name__)
2 if __name__ == "__main__":
3     print "Execution directe"
4 if __name__ == "modularite_execution":
5     print "Importation"
```

```
1 $ python modularite_execution.py
2 Nom : __main__
3 Execution directe
```

```
1 >>> import modularite_execution
2 Importation
```

## Modularité - Packages

On peut avoir un dossier complet pour module avec chaque fichier en sous-module.

Fichier obligatoire : pack/\_\_\_init\_\_\_py

```
1 print "Importation du package pack"
```

pack/test.py

```
1 def f():  
2     print "beuh !"
```

```
1 >>> import pack  
2 Importation du package pack  
3 >>> import pack.test  
4 >>> pack.test.f()  
5 beuh !
```

## Modularité - Sélection fine

Je ne veux qu'une variable, fonction ou classe

```
1 >>> from pack.text import f
2 >>> pack
3 NameError: name 'pack' is not defined
4 >>> f
5 <function f at 0x7f09052ab5f0>
```



# Tests unitaires

Un test unitaire permet de tester un programme en vérifiant généralement les retours de fonctions par rapport à des arguments données en guise de test

Des tests ? Pourquoi faire ?

- Maîtriser sa peur de codeur en vérifiant que le programme fonctionne
- Connaître rapidement l'état de dégradation de son code :P
- Ne pas faire du refactoring à l'aveuglette
- Améliorer la qualité de son programme
- Avoir un robot qui roule le jour de la coupe :D

# Nose

Utilisation du module nose

Doc officielle (en)

Didacticiel (en)

```
1 $ sudo apt-get install python-nose
```

Pourquoi Nose ?

- Le module *unittest* existe dans la librairie standard mais il lui manque quelques fonctionnalités intéressantes
- Des tests qui s'écrivent facilement et rapidement dans une syntaxe python
- Couverture de code
- Configuration fine, plugins
- Performance : plugin pour faire du multithreadé

## Utilisation basique de Nose

demo\_tests.py

```
1 import re
2 EMAIL_REGEX = r'[\S.]+@[\S.]+'
3
4 class testEmail:
5     def test_email_regex(self):
6         assert re.match(EMAIL_REGEX, 'test@mail.ru')
7         assert not re.match(EMAIL_REGEX, 'test@where')
```

## Utilisation basique de Nose

```
1 $ nosetests demo_tests.py
2 F
3
4 FAIL: demo_tests.testEmail.test_email_regex
5
6 [...]
7     assert not re.match(EMAIL_REGEX, 'test@where')
8 AssertionError
9
10
11 Ran 1 test in 0.001s
12
13 FAILED (failures=1)
```

## Téléchargeur sans threads

```
1 import urllib
2 import time
3
4 def get_file(url):
5     mytime = time.ctime()
6     f = urllib.urlopen(url)
7     contents = f.read()
8     f.close()
9     print mytime+" "+url+" OK"
10
11 time_ini = time.ctime()
12
13 monurl = "http://dl.google.com/android/android-
14 for url in [monurl]*5:
15     get_file(url)
16
17 print time_ini
18 print time.ctime()
```

```
1 Sun Dec 11 23:58:00 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
2 Sun Dec 11 23:58:05 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
3 Sun Dec 11 23:58:06 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
4 Sun Dec 11 23:58:08 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
5 Sun Dec 11 23:58:09 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
6 Sun Dec 11 23:58:00 2011
7 Sun Dec 11 23:58:11 2011
```

## Téléchargeur avec threads

```
1 import urllib , threading          15 contents = f.read()
2 import time                        16 f.close()
3                                   17 print mytime+" "+url+" OK"
4 class FileGetter(threading.Thread): 18
5     def __init__(self, url):        19 time_ini = time.ctime()
6         self.url = url              20 tab = []
7         threading.Thread.__init__(self, url) 21 monurl = " http://dl.google.com/android/android-
8
9     def get_result(self):           22 for url in [monurl]*5:
10        return self.result          23     tab.append( FileGetter(url))
11
12    def run(self):                   24     tab[len(tab)-1].start()
13        mytime = time.ctime()       25 for thread in tab:
14        f = urllib.urlopen(url)     26     thread.join()
15                                   27 print time_ini
16                                   28 print time.ctime()
```

```
1 Sun Dec 11 23:57:56 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
2 Sun Dec 11 23:57:56 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
3 Sun Dec 11 23:57:56 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
4 Sun Dec 11 23:57:56 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
5 Sun Dec 11 23:57:56 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
6 Sun Dec 11 23:57:56 2011
7 Sun Dec 11 23:58:04 2011
```

## Améliorations à apporter

De nombreux d'autres problèmes sont à gérer :

- Gérer l'accès concurrent à une même ressource
- Mettre une file d'attente (voir de priorité) pour limiter à sa guise le nombre de threads
- Utiliser le module multiprocessing pour faire du multi-cœurs
- ...

# socat

OMG J'émule une liaison série

## Le site officiel (en)

```
1 $ sudo apt-get install socat
2 $ sudo apt-get install python-serial
3 $ socat PTY,link=./ptyp1,b9600 PTY,link=./ptyp2,b9600
```

ptyp1 et ptyp2 sont des pseudo terminaux reliés virtuellement par une liaison série.



# Pyserial

Ouvrir deux autres terminaux (dans le même dossier) et lancer \$  
*sudo python*

```
1 >>> import serial
2 >>> ser = serial.Serial('./ptyp1')
3 >>> while 1:
4 ...     ser.readline()
5 ...
6 'INTech va tout faire peter !\n'
```

```
7
8 >>>import serial
9 >>>ser = serial.Serial('./ptyp2')
10 >>>ser.write("INTech va tout faire peter !\n")
11 29
```

```
1 >>> # debit de Baud de 19200, timeout 1s
2 >>> ser = serial.Serial('./ptyp1', 19200, timeout=1)
3 >>> ser.read() # Lire un bit
4 >>> ser.read(10) # Lire 10 bits
5 >>> ser.close # fermer la liaison proprement
```

## Présentation de PyPy

PyPy est une implémentation de python (2.7.1 pour l'instant) destinée à accélérer, en moyenne 3x plus vite, l'exécution d'un programme en python (**Attention** : il faut tester au cas par cas)  
C'est un programme python (!!)

qui fait de la compilation à la volée (JIT Compilation : Just-In-Time Compilation)

### Télécharger PyPy

Choisir "JIT Compiler" version

Dézipper le tout, le programme est dans pypy-1.7/bin/pypy

Créer un lien symbolique de pypy (avec le chemin absolu) vers  
/usr/bin

## Pour moi qui code, quelles sont les différences ?

- lancer pypy et non python
- Sauf pour des extensions exotiques, persos, ... les principales fonctionnalités sont gérées à l'identique
- Sur ma distribution Linux, mon Mac OS X tout va bien. Sur Windows, c'est un peu moins stable.
- Ce qui **ne marche pas** : trac, Zope, Genshi, matplotlib :(  
NumPy (en cours) :(  
gmpy, SciPy :(  
SAGE, Bazaar, Subversion,  
SCons, pygame :(  
pyGTK, Tkinter, Cython, Pyrex, SWIG, py2exe,  
Fabric, Pyro
- Ce qui **marche** : Django, Pylons, Pyramid, Twisted, IPython, Scapy :), SQLAlchemy, pymongo, PyMySQL, pysqlite, redis, PIL, mechanize :), mock, nose :), Selenium Python Client Driver, unittest2

# Print

## doc

```
1 >>> print "The answer is" , 42 # Python 2.x
2 The answer is 42
3 >>> print("The answer is" , 42) # Python 3.x
4 The answer is 42
5 >>> print(1,2) # Python 2.x
6 (1,2)
7 >>> print(1,2) # Python 3.x
8 1 2
```

## Les Dictionnaires

```
1 >>># Python 2.x
2 >>>dict = {(1,2): "truc"}
3 >>>dict.keys()
4 [(1, 2)]
5
6 >>> # Python 3.x
7 >>> dict = {(1,2): "truc"}
8 >>> dict.keys()
9 dict_keys([(1, 2)])
10 >>> list(dict.keys())
11 [(1, 2)]
```

Idem pour *dict.items()*, *dict.values()*

## map()

```
1 >>> # Python 2.x
2 >>> map(lambda x: x**2, [1, 2])
3 [1, 4]
4
5 >>> # Python 3.x
6 >>> map(lambda x: x**2, [1, 2])
7 <map object at 0x2327590>
8 >>> list(map(lambda x: x**2, [1, 2]))
9 [1, 4]
```

Beaucoup d'autres changements moins importants.

Globalement, Python 3.x est plus orienté objet.

Des changements introduits dans Python 3.x sont peu à peu introduits dans Python 2.x avec rétrocompatibilité.

## Listes compréhensives

```
1 >>> [n ** 2 for n in range(10)]
2 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
3 >>> [n ** 2 for n in range(10) if n % 2]
4 [1, 9, 25, 49, 81]
```

## Ce qui n'a pas été abordé

Voir la superbe présentation de Boris Feld

- Gestion des exceptions
- Les ensembles
- \*args et \*\*kwargs
- Callbacks
- Fonctions anonymes/lambda (entrevu)
- Fonction filter
- Les générateurs (le robot en aura peut-être besoin, à voir)
- Les décorateurs



## Ce qui n'a pas été abordé

Voir sur le Net ou venir me voir

- Notions intermédiaires et avancée de la programmation orientée objet
- Gestion du verouillage, des files d'attente/de priorité pour les threads
- Utilisation intermédiaire et avancée de Nose
- Programmation avec les sockets (mais y'avait un club code dessus :P)

# Licence



Cette oeuvre est mise à disposition selon les termes de la  
Licence Creative Commons Paternité -  
Partage à l'Identique 3.0 non transposé.